Effective Switch Memory Management in OpenFlow Networks

Anilkumar Vishnoi, Rishabh Poddar, Vijay Mann, Suparna Bhattacharya
{avishnoi, rishpodd, vijamann, bsuparna}@in.ibm.com
IBM Research, India

Abstract

OpenFlow networks require installation of flow rules in a limited capacity switch memory (mainly Ternary Content Addressable Memory or TCAMs) from a logically centralized controller. A controller can manage the switch memory in an OpenFlow network through events that are generated by the switch at discrete time intervals. Recent studies have shown that data centers can have up to 10,000 network flows per second per server rack today. Increasing the TCAM size to accommodate these large number of flow rules is not a viable solution since TCAM is costly and power hungry. Current OpenFlow controllers handle this issue by installing flow rules with a default idle timeout after which the switch automatically evicts the rule from its TCAM. This results in inefficient usage of switch memory for short lived flows when the timeout is too high and in increased controller workload for frequent flows when the timeout is too low.

In this context, we present SmartTime - an OpenFlow controller system that combines an adaptive timeout heuristic to compute efficient idle timeouts with proactive eviction of flow rules, which results in effective utilization of TCAM space while ensuring that TCAM misses (or controller load) does not increase. To the best of our knowledge, SmartTime is the first real implementation of an intelligent flow management strategy in an OpenFlow controller that can be deployed in current OpenFlow networks. In our experiments using multiple real data center packet traces and cache sizes, SmartTime adaptive policy consistently outperformed the best performing static idle timeout policy or random eviction policy by up to 58% in terms of total cost.

Categories and Subject Descriptors: C.2.4 [Distributed Systems]: Network operating systems

General Terms: Performance

Keywords: OpenFlow, idle timeout, Software Defined Networking

1. INTRODUCTION

Software defined networking using protocols such as OpenFlow [19] is quickly gaining popularity and adoption in modern data centers [16, 17]. OpenFlow provides flexibility through programmable route computation as the control plane is physically decoupled from all forwarding switches (the data plane). However, this flexibility comes at the cost of placing significant stress on switch state size as OpenFlow requires installation of flow rules in a limited capacity switch memory (mainly Ternary Content Addressable Memory or TCAMs). Most commercial OpenFlow switches have an on-chip TCAM with a size that accommodates between 750 to 2000 OpenFlow rules [4, 10]. The state of the art Broadcom chipset, which is used by most commercial switches today, has a TCAM that accommodates 2000 OpenFlow rules [15]. Recent studies have shown that data centers can have up to 10,000 network flows per second per server rack today [22]. Increasing the TCAM size to accommodate flow rules for these large number of flows is not a viable solution since TCAM is costly and power hungry.

To overcome the problem of limited switch memory, OpenFlow specification allows each OpenFlow flow rule to have an idle timeout period which controls its eviction from the switch memory. If no packet matches a given flow rule for a period equal to its idle timeout period, the flow rule is removed from the switch memory. This idle timeout is set by the controller before it sends a flow rule to the switch for installation.

However, in the absence of an intelligent way to figure out what flows are frequent or short lived, a controller usually installs flow rules with a default idle timeout value (usually 5 seconds) or the minimum timeout value specified in the current OpenFlow specifications (1 second) or infinite timeout. Furthermore, almost all the current OpenFlow controller implementations [3, 8, 12] install flow rules for all flows with the same idle timeout. Studies show that data center flows vary widely in their duration with almost 50% of flows being less than 1 second duration and 80% of the flows being less than 10 seconds [21]. This results in inefficient usage of switch memory for short lived flows when the timeout is too high and in increased controller workload for frequent flows when the timeout is too low.

There has been little work on dynamic timeout assignment and switch memory management approach that will work in real OpenFlow networks. Given the extremely small TCAM size in most commercial OpenFlow switches, and TCAM being an extremely costly resource, this problem needs to be addressed to ensure widespread deployment of OpenFlow switches in production networks. A controller can manage the switch memory in an OpenFlow network through events that are generated by the switch at discrete time intervals. This includes the “packet-in" event on a cache miss and a “flow removal" event on a flow rule expiry. In this context, we present SmartTime - an OpenFlow controller system that employs an adaptive heuristic to compute idle timeouts for flow rules which results in effective utilization of TCAM space while ensuring that the misses (or controller load) reduce as compared to a baseline static policy. It takes into account the current TCAM utilization level and the likelihood of a flow appearing again in the

DEBS'14, May 26-29, 2014,Mumbai, India

network while deciding the flow rule idle timeout. It also leverages proactive eviction of flow rules in case the TCAM is close to its full capacity. Specifically, we make the following contributions in this paper:

C1: Design and implementation of SmartTime in two popular OpenFlow controllers: Floodlight [3] and OpenDaylight [8]. To the best of our knowledge, SmartTime is the first real implementation of an OpenFlow controller system that combines adaptive idle timeouts for flow rules with a proactive eviction strategy based on current TCAM utilization level and can be deployed in current OpenFlow networks.

C2: Adaptive idle timeout heuristic based on analysis of real data center packet traces: We analyze real data center packet traces and design our adaptive strategy based on some key observations. We observe that many flows in the network never repeat, and the current minimum timeout value of 1 second for OpenFlow flow rules is too large for such flows. We recommend a smaller minimum timeout value for OpenFlow in the range of 10-100 milliseconds.

C3: Validation of SmartTime using real data center packet traces: We validated SmartTime by replaying four real data center packet traces for two representative cache sizes and compare it with multiple static idle timeout policies and random eviction policies. In all our experiments, SmartTime adaptive policy was either the best performing or second best performing policy across all traces and cache sizes. In 67 out of 72 experiments, SmartTime adaptive policy outperformed the best performing static idle timeout policy or random eviction policy by up to 58% in terms of total cost.

The rest of this paper is organized as follows. Section 2 gives an overview of related research. In section 3 we present a formal description of the problem of predicting idle timeout for flow rules. Section 4 presents our analysis of data center network traces, which forms the basis of our adaptive idle timeout heuristic. In section 5, we describe our implementation of SmartTime using the Floodlight OpenFlow controller. We provide an experimental evaluation of SmartTime in Section 6. Finally, we conclude the paper in Section 7.

2. RELATED WORK

While much research attention has been devoted to compacting the representation of flow rules in order to reduce TCAM space [13], the problem of choosing optimal timeouts for flow rules in a real OpenFlow network remains largely under-explored so far.

Zarek et al. [23] have observed, using simulations, that TCAM miss rates do not improve significantly beyond a certain timeout value and that this timeout value is different for different traces, which corroborates our findings on the need for an adaptive strategy. They proposed combining fixed uniform idle timeout values (across all flow rules) with proactive eviction messages from the controller. However, their implementation of the strategy requires detection of TCP flow completion by matching on TCP header flags (SYN, RST, FIN) and such matching is not supported in current OpenFlow switch implementations that support OpenFlow 1.0 specification. They also explore the use of random and FIFO eviction policies, which are implementable but less effective in the absence of a mechanism to take into account the differing characteristics of different flows.

Ryu et al. [20] presented an adaptive strategy called MBET (Measurement Based Binary Exponential Timeout) for timing out Internet flows in a router. Their approach relies on persistency in intra-flow packet inter-arrival times. It assigns a high timeout to each new flow and then successively reduces it as long as the measured throughput exceeds a given threshold. A similar solution has also been proposed in [14] to examine flows in an SDN switch and predict if and when they are likely to be evicted. Both these solutions have been implemented in simulators. Unlike our proposed adaptive heuristic, they can not be implemented in current OpenFlow networks because of the following reasons:

R1: OpenFlow specification does not allow altering the idle timeout of a flow rule already installed in a switch.

R2: Even if OpenFlow allowed changes to the idle timeout of an existing flow rule, these strategies will require extensive polling by the controller which can be prohibitive even in a medium sized data center.

3. PROBLEM DESCRIPTION AND FORMULATION

In this section we formulate the problem of minimizing cost of a TCAM miss in an OpenFlow network. Figure 1 illustrates the importance of choosing the right idle timeout when installing a flow rule in a switch. If the timeout value is too high for *a given flow rule*, the flow rule sits in the switch TCAM for a long time, wasting valuable TCAM space, which could be utilized by *another flow rule*. Such large timeouts result in high TCAM utilization and may eventually lead to packet drops when the TCAM becomes full. In order to avoid such packet drops, proactive eviction of flow rules that are chosen either randomly or in FIFO order by the controller has been proposed in earlier research [23].

On the other hand, if the timeout value is too low for a given flow rule, the flow rule gets expired from the switch too quickly. As per OpenFlow specification, any subsequent packet that would have matched this flow rule, results in a TCAM miss and gets redirected to the OpenFlow controller. The OpenFlow controller will then install an appropriate routing flow rule (as per its forwarding policy implementation) in the switch which gets applied to the packet that was sent to controller as well as to subsequent packets matching that flow rule. However, this additional round trip to the controller is costly in terms of latency (for the few initial packets of a flow). Authors in [10] point out that while switch latencies are in microseconds, a single round trip to controller results in an additional latency of around 10-20 milliseconds. Furthermore, this additional round trip also increases the controller workload, which can prove to be a bottleneck in a large data center.

Assuming that during a short time interval of duration T, for a TCAM of size S:

F	Number of flows
$N(f)$	Number of packets for flow f = $N(f,T)$
$IA(f)$	Interarrival time for flow f
$IT(f)$	Idle timeout for flow f
U_{max}	Maximum fraction of TCAM utilization desired (allowing headroom for flow bursts)
$E_{evicted}(f)$	Event wherein a flow f is evicted from the TCAM
$E_{expired}(f)$	Event wherein a flow f naturally expires i.e. $IA(f) > IT(f)$
$P_{miss}(f)$	Probability that there is a miss in the switch TCAM for flow f
$U_{held}(f)$	= 1, if flow f is held in TCAM (i.e. its flow rule has not expired or been evicted) = 0, otherwise
F_{held}	Number of flow (rules) held in the TCAM $= \sum_f U_{held}(f)$
M_{avg}	Average number of misses in switch TCAM
P_{evict}	Probability that a miss in the switch TCAM will lead to the eviction of some installed flow, e.g. if the TCAM is full $= P(F_{held} \geq U_{max})$
M_{evict}	Average number of evictions in the switch TCAM $= P_{evict} M_{avg}$
$C_{install}$	Cost of installing a flow in the switch TCAM in the event of a miss (including the overhead of an additional round trip to the controller)
C_{evict}	Additional overhead of evicting a flow from the switch TCAM before an installation
C_{miss}	Cost of a miss in the switch TCAM

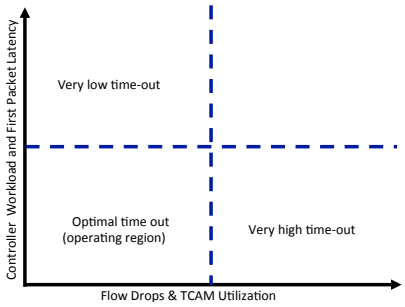


Figure 1: A low idle timeout will result in higher TCAM miss (higher controller workload and packet latencies), whereas a high idle timeout will result in higher TCAM utilization and flow drops (or proactive flow evictions)

Our Problem:

- Objective: Choose $IT(f)$ to reduce average cost[1] incurred in every interval T:

$$C_{avg} = C_{miss} M_{avg} \tag{1}$$

Note that the cost of a miss is equal to the cost associated with an additional round trip to the controller and the cost of installing a new flow rule in the switch TCAM. Further, the controller may decide to optionally evict an existing flow rule, in case of some misses (for example, this could be done when the TCAM utilization is 100% or it crosses a particular threshold). Therefore,

$$C_{miss} = C_{install} + P_{evict} C_{evict} \tag{2}$$

Average cost can then be represented as:

$$C_{avg} = (C_{install} + P_{evict} C_{evict}) M_{avg} \tag{3}$$
$$= C_{install} M_{avg} + C_{evict} M_{evict} \tag{4}$$

where average misses is computed as follows:

$$M_{avg} = \frac{1}{\sum_f N(f)} \sum_f N(f) P_{miss}(f) \tag{5}$$

For a flow f, a miss can occur either if its flow rule naturally expired (interarrival time exceeded idle timeout), or if it was forcefully evicted from the switch TCAM by the controller (for example, to accommodate some other flow, as described above). Thus,

$$P_{miss}(f) = P(E_{expired}(f) \vee E_{evicted}(f)) \tag{6}$$
$$= P(IA(f) > IT(f) \vee E_{evicted}(f)) \tag{7}$$

and average misses can thus be represented as

$$M_{avg} = \frac{1}{\sum_f N(f)} \sum_f N(f) [P(IA(f) > IT(f) \vee E_{evicted}(f)] \tag{8}$$

- Subject to: TCAM size (utilization) constraint

$$\frac{F_{held}}{S} \leq U_{max} \tag{9}$$

[1]We do not consider the cost of hits here as they are characteristic of the network elements and are unavoidable

where,

$$E[F_{held}] = \sum_f E[U_{held}(f)]$$
$$= \sum_f 1 - P_{miss}(f)$$
$$= F - \sum_f P(IA(f) > IT(f) \vee E_{evicted}(f) \tag{10}$$

3.1 Why the solution is non-trivial

The above formulation indicates that we should choose $IT(f)$ such that:

- high packet rate flows (high $N(f)$ in a given interval T) have a very low miss probability in order to keep average misses down (from Equation 8), while
- low packet rate flows have a very high likelihood of misses, to help meet the TCAM space constraint (Equations 8, 9 and 10).

In the extreme case, we could set $IT(f) \rightarrow 0$ (or the minimum supported timeout) for the K lowest packet rate flows observed (where K is chosen to meet the TCAM size constraint, e.g. $K \geq F - U_{max} S$) and set $IT(f) \rightarrow \infty$ (or the maximum supported timeout) for all the remaining (higher packet rates) flows in order to minimize misses.

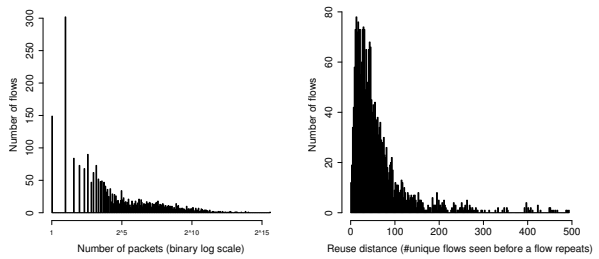
However, any static policy, however optimal in a given interval, does not adapt well to the changing dynamics of packet flows, we have observed. For example, if a flow is assigned a high timeout based on the above logic but is no longer frequent, it still continues to occupy TCAM space as the controller does not get notified until the timeout expires. Further, if a flow is so frequent that its rule never has a miss in the switch, then the controller does not get any notifications either, so it cannot distinguish this flow from those that are no longer frequent.

Hence, for a policy to be responsive to changes in traffic (esp bursty flows) it needs to adapt to actual inter-arrival times more smoothly and result in rule evictions once in a while even for high data rate packets. As a result, curiously enough, we have to strike a balance between making timeouts a non-decreasing function of inter-arrival times for optimality reasons versus making them proportional to inter-arrival times to ensure responsiveness.

Moreover, in conjunction with adaptive timeouts for flows, proactive evictions can be employed usefully to avoid packet drops (as proposed in [23]), which would otherwise lead to repeated misses in the switch TCAM until space becomes available for installation of the flow. To keep average misses down, these evictions should be as infrequent as possible (from Equation 8), which is easily achieved by evicting only when the TCAM crosses an acceptable threshold. Additionally, the flow to be evicted needs to be selected carefully, and should ideally be one that has the lowest likelihood of being observed in the near future, and hence expected to have naturally timed out before the arrival of the next packet. The eviction of such a flow would serve to free up valuable TCAM space well in advance without adversely impacting the number of misses. However, as described earlier, it may not always be possible to distinguish such a flow from a frequent one, and leading to evictions for high data rate packets once in a while.

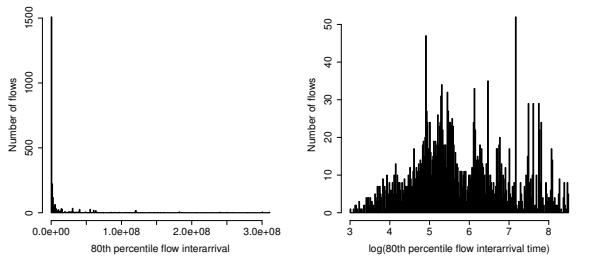
In summary, the overall problem becomes challenging to solve in real data centers because of the following reasons:

- First, all the parameters for every flow are not known upfront.



(a) Number of packets across flows (b) Reuse distance histogram

Figure 2: Packet trace analysis



(a) 80th percentile (b) 80th percentile on log scale

Figure 3: Flow inter-arrival times: x-axis is in microseconds

- Second, the parameters are likely to vary dynamically (phases and bursts can occur), and flows come and go. A static (point-in-time) optimization is unlikely to be responsive to these changes.

- Third, at the controller, we only get indication about a flow's dynamics or parameters on a miss.

To gain more insight on these practical considerations, we next study flow characteristics of publicly available data center network traces.

4. TRACE ANALYSIS

We analyze real packet traces from university data centers used in [22] and publicly available at [1]. These data centers serve the students and administrative staff of the university and provide a variety of services such as system back-ups, hosting distributed file systems, E-mail servers, Web services (administrative sites and web portals) and multicast video streams.

Our analysis treats packets between a given source and destination pair as belonging to the same flow (rule) irrespective of their inter-arrival times. As can be observed from Fig 2(a), there is a wide variation in the number of packets across flows. About 15% of flows have just 1-2 packets. To avoid wasting TCAM space, the idle timeout should be set to a very small value for such flows. This corroborates a similar finding previously reported by Ryu et al. [20] in the general context of adaptive timeouts for detecting Internet flows. The authors of [20] classified flows as small (1-2 packets), medium (between 2-10 packets) and large (greater than 10 packets). They also noted a high variability in flow duration and size (spanning 4 orders of magnitude).

4.1 Reuse distance analysis

In traditional cache optimization, a reuse distance analysis is often used to characterize memory access locality of a workload (program) in a hardware independent manner. The reuse distance (also known as LRU stack distance) is the number of distinct elements referenced between consecutive references to the same data [11, 18]. We adopt a similar approach in keeping with our view of the switch TCAM memory as a cache for flow rules (which must be retrieved from the controller on a miss). The reuse distance of a flow is defined as the number of distinct flows arriving at the switch between two consecutive packets matching the flow. Note that unlike reuse distance in program locality analysis, the workload of flows processed by the switch does not correspond to a single application or source. It just characterizes a given network trace of traffic at the switch.

Fig 2(b) is a histogram of the average reuse distance of the flows seen in the trace after filtering out flows that do not repeat. We observe that a large percentage of flows that repeat seem to have a reuse distance within 150, which is much smaller than the size of the switch TCAM. This indicates that the workload is well suited for an LRU replacement policy. However, although the effectiveness of an LRU policy has previously been empirically confirmed using simulations, directly implementing LRU in an OpenFlow switch is impractical.

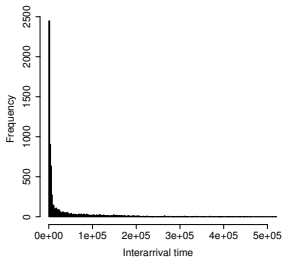
Why LRU is not suitable: Any LRU algorithm would require real-time information about the usage pattern of a cache entry. An OpenFlow enabled switch has this usage information and can, in theory, implement an LRU eviction strategy. However, one of the founding principles of OpenFlow, has been to delegate all intelligence and control functions to the OpenFlow controller (the control plane). This enables the switch to be manufactured out of cheap commodity hardware (ASICs). Therefore, OpenFlow specification forbids installation or removal of a flow rule entry by the switch itself. Implementing an LRU policy at the OpenFlow controller is also infeasible since the controller can not fetch real-time information about the usage pattern of a flow rule entry that is currently installed in a switch. A controller can query the switch periodically to find out the active time of various flow entries in the switch. However, the active time of a flow entry in the switch includes the time for which it has been idle and hence can not be used to implement an LRU eviction strategy. Furthermore, this polling, can have significant overheads and is unlikely to scale even for medium sized data centers.

Could an LRU-like effect can be achieved by dynamically predicting idle timeouts for flows at the controller? To help explore this possibility, we next analyze interarrival times across flows.

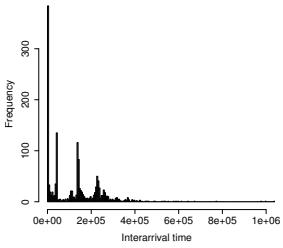
4.2 Flow interarrival times (IAT) across flows

We record interarrival times for packets corresponding to a flow when they are more than 1 ms apart. 1 ms is chosen as the flow boundary to ensure that we consider 1 ms and above as legitimate idle timeout values. Here are a few observations based on the results of the analysis:

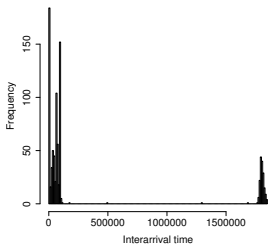
The 80th percentile of per flow interarrival times spans a wide range: For each flow we determine the 80th percentile of its interarrival times (i.e. the lowest setting of idle timeout at which 80% of the packets for the flow arrive before the flow rule times out). 80% is chosen to ensure that a wide majority of the packets for a flow match the corresponding flow rule without resulting in a miss. Packets with interarrival times higher than 80th percentile may represent packets that repeat after a long a gap and setting an idle timeout equal to these values may result in a flow rule sitting idle in the TCAM for too long. The histogram of these values (refer Figure 3(a)) shows a good number of flows with small interarrival



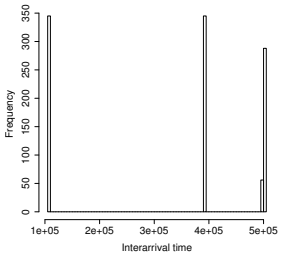
(a) Flow rule 1722 - low IATs



(b) Flow rule 1916 - IATs spread uniformly in clusters till 500 ms



(c) Flow rule 74 - clusters of IATs far apart



(d) Flow rule 2135 - discrete IATs far apart

Figure 4: Differences in the magnitude and shape of the histogram of flow interarrival times across flows - x-axis is in microseconds

times (below 500ms), but the long tail indicates that the interarrival times vary over a very wide range for the rest of the flows (spanning multiple orders of magnitude, going all the way up to 300 seconds). This can be seen more clearly in Figure 3(b)where the same histogram is plotted against a log scale for the interarrival times.

Even the shape of interarrival time histograms can be very different for different flows: Fig 4 illustrates the histogram of interarrival times corresponding to four sample flow rules which match a large number of packets in the observation interval for which the trace was collected. Note the difference in the patterns for 4 different rules.

4.3 Number of cache misses across flows

We capture the number of misses experienced by each flow at the controller, when run with a fixed idle timeout policy of 5 seconds and a TCAM size of 750. We used four different traces for this analysis, described in greater detail in Section 6 – EDU1, EDU2, SMIA1 and SMIA2. To do this, we used the experimental setup shown in Figure 9 and later described in Section 6. The results of this analysis are presented in Figure 5, after indexing the flows in increasing order of number of misses. Across all the traces, we observe that less than 20% of the flows contribute towards over 80% of the total misses. This disparity is especially pronounced in SMIA1 and SMIA2, with over 90% of the total misses being the result of less than 10% of the flows. This observation leads us to conclude that most misses are caused by only a few flows, and having a single idle timeout across all flows is inefficient.

4.4 Inferences

Together our observations confirm that:

- a lot of flows have really low inter-arrival times while some flows hardly repeat - a minimum idle timeout of 1 second is too large for both these types of flows
- assigning a single (static or fixed) idle timeout for all flows would be inefficient as most of the misses are caused by a few flows
- even the per-flow idle timeout should be assigned dynamically and adapt to changes in traffic patterns

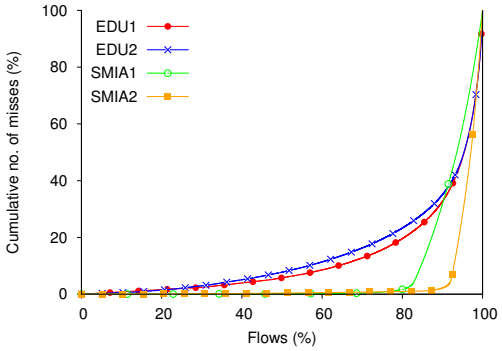


Figure 5: Number of misses across flows: y-axis represents the cumulative no. of misses as a % of the total no. of misses; x-axis represents % of flows

5. DESIGN AND IMPLEMENTATION

In this section we describe our design and implementation of SmartTime including our adaptive idle timeout strategies.

5.1 Design

We now present our adaptive idle timeout strategy based on the analytical formulation presented in Section 3 and our findings in Section 4. Using the problem formulation presented in Section 3, we can derive some key guidelines for our strategy as follows.

- As discussed in Section 3 we should set the lowest possible timeout for flows that either have low data rates or are known not to repeat. Prior studies [21] have reported that most (50%) data center flows last less than 1 second. Figure 3(a) reconfirms that the 80th percentile of inter-arrival times for the majority of flows is less than 1 second.
- In equation 5 the term $N(f) * P_{miss}(f)$ denotes the number of misses for a given flow f, which is the same as the flow repeat count for f observed at the controller. Making the idle timeout a steeply increasing function of the repeat count observed can reduce the contribution of this term for flows which repeatedly incur misses.
- However, a larger timeout implies a large delay before the controller gets notified in case activity for the flow slows down. Hence, a careful eviction policy (effectively a forced early timeout) is needed to continue meeting the constraint in equation 9 as new flows become active, while ensuring that the number of misses are not adversely impacted due to the eviction of a high packet rate flow.

The pseudocode for our SmartTime adaptive strategy (Adaptive-R) is given in Figure 6. We now describe its key features.

F1: Small initial idle timeout. Our adaptive schemes assign all flows to start with a low idle timeout of 100 milliseconds (line 4 in Figure 6). This ensures that short flows and flows that never repeat do not sit in TCAM for long. This is critical since most of the flows observed in our analysis were short-lived and never caused

```
1: function GetIdleTime
2:     FlowRepeatCount + =1
3:     if a flow has never been observed before then
4:         IdleTimeout = MinIdleTimeout (100 ms)
5:     else
6:         IdleTimeout = MinIdleTimeout * 2^FlowRepeatCount
7:         if FlowPrevIdleTimeout == MaxIdleTimeout AND AvgHoldFactor > 3
           then
8:             IdleTimeout = MinIdleTimeout (100 ms)
9:         if FlowPrevIdleTimeout == MaxIdleTimeout then
10:            IdleTimeout = MaxIdleTimeout (10 s)
11:        if TCAMUtilization > 95% then
12:            Evict a flow rule randomly
13:    FlowPrevIdleTimeout=IdleTimeout
14:    return IdleTimeout
```

Figure 6: Pseudocode for SmartTime Adaptive Strategy (Adaptive-R)

any misses. This timeout was determined empirically by experimenting with other smaller timeouts (1ms and 10ms). A smaller timeout than 100ms results in higher misses while keeping the evictions low. A higher timeout will results in a huge increase in evictions while marginally reducing the misses. 100ms represents a good balance between the number of misses and proactive evictions. Further, this is a tunable parameter, and if one wants to lower the number of evictions (perhaps because of a high cost of eviction, which will govern the average cost as per Equation 4 in Section 3), one can start with a lower initial timeout such as as 10ms. Since, the minimum idle timeout supported by current OpenFlow protocols is 1 second, we modified OpenvSwitch [7] to accommodate idle timeout values in milliseconds.

F2: Rapid ramp up from small timeout for frequent flows. After selecting a small value for initial idle timeout, it is important that we "ramp-up" as quickly as possible for flows that repeat often. We ensure that by exponentially increasing the timeout on each repeat occurrence (line 6 in Figure 6).

F3: Cap on maximum idle timeout. Exponential increase in timeouts need to be limited after a few occurrences so that we do not increase the idle timeout to very high values which can result in wasted TCAM space in case of mispredictions. As 80% of the flows are reported to be less than 10 seconds [21], we bound the maximum idle timeout at 10 seconds (line 10 in Figure 6).

F4: Timeout reduction for short flows that repeat often but after a long gap. Our analysis in Section 4 shows that some flows can have widely varying inter-arrival times (refer Figures 4(c) and 4(d)) and some inter-arrival times may be large. This implies that such flows should not have the benefit of a large idle timeout when they repeat after a long time and when they are likely to last for a short duration. Our adaptive schemes achieve this by reducing the idle timeout for flows to minimum timeout (line 8 in Figure 6)when these flows continue to have a bad average hold factor [20][2] (greater than 3) even after they have repeated several times (i.e reached the max idle timeout of 10 seconds). While, this keeps our heuristic simple, it can result in the controller reacting slowly to inactive flows. We are currently working on further fine tuning our adaptive policy so that it can react to inactive flows faster by taking into account feedback on number of packets or bytes that have matched an expired flow rule. Controller receives this information as part of the "flow removal" event.

[2]Hold Factor is defined as the sum of active time and idle time, divided by the active time: it is desirable to have the hold factor as close to 1 as possible

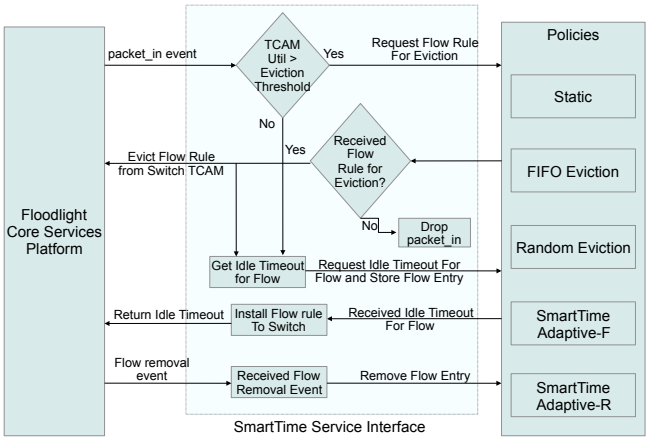


Figure 7: SmartTime Implementation in Floodlight OpenFlow Controller

F5: Proactive eviction for flows when TCAM is about to get full. When TCAM utilization crosses an eviction threshold (currently defined as 95%), adaptive strategies start evicting flow rules in a random manner (line 12 in Figure 6). Ideally these rules should be evicted by a background thread to avoid increase in flow setup latencies. Victims for eviction can also be chosen based on bad average data rate or bad hold factor. We also experimented with using FIFO order for rule removal as discussed in related research [23]. Our experiments confirmed that random eviction always performs better than FIFO eviction across all the traces. This can be attributed to the fact that a large majority of the misses are caused by a few (flows (as shown in Figure 5. Hence, the probability of evicting a less popular flow (or a flow with lower frequency) is higher if it is chosen randomly rather than using FIFO order.

5.2 Implementation

Figure 7 shows the SmartTime architecture as implemented in Floodlight OpenFlow Controller. SmartTime service exposes an interface for various adaptive, static and proactive eviction polices to be implemented. SmartTime module registers itself for 2 OpenFlow events from the Floodlight infrastructure:

packet-in event: Switch sends this event to the controller whenever a new flow arrives at the switch and there is no matching flow rule (first few bytes of the first packet are also sent to the controller). This event may also be sent by another module (e.g. the default forwarding module in Floodlight that needs to compute an idle timeout for the flow rules it installs).

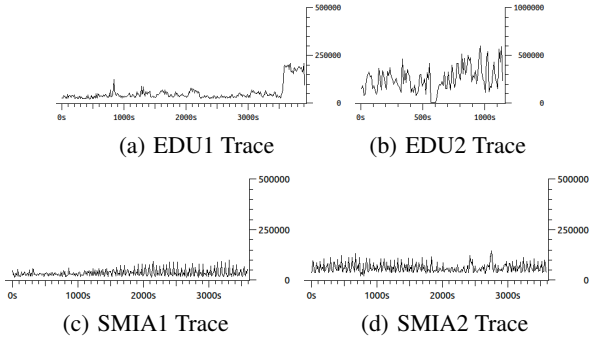
flow removal event: Switch sends this event to the controller whenever a flow rule is removed from the switch either as a result of an idle timeout or proactive forced eviction by the controller;

In addition, the SmartTime controller module also fetches the maximum TCAM size associated with each switch whenever a switch connects to the controller (using the "OFPST_TABLE" OpenFlow request).

Once a packet-in event is received by the SmartTime service, it checks if the current TCAM utilization level is less than the eviction threshold (currently configured to a static value of 95%, we plan to modify it later based on arrival flow rate). If the utilization level is less than the eviction threshold, the SmartTime service fetches the idle timeout from the currently active timeout policy - we currently support static, FIFO proactive eviction, Random proactive eviction and two Adaptive policies (with FIFO and Random evictions once TCAM utilization crosses eviction threshold). SmartTime service

S.No.	Trace	Duration (mins)	Packets/sec	Unique flows
1	EDU1	65	5072	14039
2	EDU2	19	10396	19319
3	SMIA1	60	3189	8744
4	SMIA2	60	5702	19475

Table 1: Summary of trace statistics



(a) EDU1 Trace (b) EDU2 Trace

(c) SMIA1 Trace (d) SMIA2 Trace

Figure 8: Packet trace plots: y-axis represents number of packets, x-axis represents time in seconds

then returns the idle timeout value to the module that invoked it. If the utilization level is more than the eviction threshold, the SmartTime service first instructs the various policies to evict a rule from the switch TCAM. Various policies decide on the flow rule(s) to be evicted based on their inbuilt eviction strategy (static policies do not have any eviction strategy by definition). Flow rules may be evicted in response to packet-in events (i.e. eviction is done in the critical path of flow rule installation) by the SmartTime service itself or by a background thread asynchronously that wakes up at regular intervals. At a later point of time when the flow rule expires from the switch, the SmartTime service also gets a flow removal event. SmartTime service passes this event to the various policies that retrieve statistics such as the total time the flow rule was active in the switch, number of bytes and packets that matched the flow rule during its life time, and update their internal data structures.

6. EXPERIMENTAL EVALUATION

We conducted two sets of experiments to evaluate SmartTime: 1) experiments to evaluate SmartTime cache performance in terms of total number of misses and drops and 2) experiments to evaluate SmartTime latency overheads.

Packet traces and replay: We used four traces in the first set of experiments: EDU1, EDU2, SMIA1 and SMIA2; EDU1 and EDU2 were captured from university data centers by the authors in [22], while SMIA1 and SMIA2 are open data sets made available by the Swedish Defense Research Agency [2]. A plot of both these traces is given in Figure 8 with number of packets on the y-axis and time on x-axis, while Table 1 summarizes the length and packets/sec for each trace.

In order to replay these packet capture traces in real-time, we used tcpreplay [9] with arguments –timer=rdtsc and –quiet. The timer flag ensures that tcpreplay uses the timestamp counter on the CPU chip for introducing timing gaps between various packets as per the captured trace. The quiet flag ensures that there are no prints executed during the replay period. We found that these flags were important in order to replay the captured trace at the same speed and with the same inter-arrival gaps.

We now describe our experiments in the next two subsections.

6.1 Evaluation of SmartTime Cache Performance

Our first set of experiments evaluates the effect of SmartTime in reducing TCAM misses (which has a direct impact in the form of reducing controller load and first packet latencies of new flows), flow drops (due to TCAM being full) or forced eviction of flow rules (in policies that involve proactive eviction of flow rules).

Testbed: Figure 9 shows the two testbeds we used for our experiments. The testbed on the left uses a modified OpenvSwitch (OVS) [7] to accommodate idle timeouts less than 1 second (which is currently the minimum idle timeout specified by the OpenFlow protocol). We modified OVS code to include idle timeouts in milliseconds. This was done to implement our adaptive strategies which ensure that all flows start with a low minimum timeout (100 milliseconds) so that the large number of flows that never repeat are evicted from the switch as soon as possible. Since OVS does not really have a TCAM, this allowed us to model a TCAM of a configurable size. We used a TCAM size of 750, since it represents the smallest TCAM size that we have come across in an enterprise-grade Top-of-Rack switch with OpenFlow support (IBM G8264 switch [4]) and it also ensured that there was sufficient contention for TCAM space across all the four traces. We also conduced experiments with TCAM size of 2000, as 2000 is a representative size for most TCAMs in current switches since Broadcom chipset used by most switches has a similar TCAM [15]. However, we found that EDU1 and SMIA1 traces were unable to populate the TCAM completely at this size as they see a lesser number of flows and have a relatively lower packets/sec rate. The other two traces - EDU2 and SMIA2, both have a larger number of flows and higher packets/sec rate, and there was sufficient contention for TCAM space even at a TCAM size of 2000. A large 12-core server replays the pcap trace using tcpreplay and SmartTime runs as part of the Floodlight OpenFlow controller on another large 12-core server. These two machines are interconnected by a single IBM G8264 [4] switch running in non-OpenFlow mode (since the OpenFlow functionality is provided by the OVS).

We now describe the 3 sets of idle timeout policies that we compare in our experiments.

P1: Static idle timeout policies: As the name suggests these policies use a constant idle timeout for all flows. We experiment with 5 different timeout values - 100ms, 500ms, 1000ms, 5000ms, and 10000ms. With these policies, when the TCAM becomes full, and a packet arrives at the switch that does not match any of the existing flows, it is sent to the controller and the controller drops that packet. Subsequent packets belonging to that flow are also dropped.

P2: Random eviction policy with static idle timeout: These policies use a constant idle timeout and behave in identical fashion to static idle timeout policies (P1 above) till the TCAM utilization reaches a pre-determined threshold (we used 95% as the threshold). When the TCAM utilization reaches 95% and a TCAM miss occurs, these policies proactively evict an existing rule. The rule to be removed is chosen at random. We also experimented with using FIFO order for rule removal as discussed in related research [23]. As discussed in Section 5, our experiments confirmed that random eviction always performs better than FIFO eviction across all the traces. We experimented with 4 different timeouts - 500ms 1000ms, 5000ms and 10000ms. We did not experiment with 100ms since our experiments with static idle timeout of 100ms showed that with 100ms idle timeout, there were hardly any drops across all the four traces (and hence there was no need to evict any rules proactively).

P3: Adaptive policy with random eviction: These policies imple-

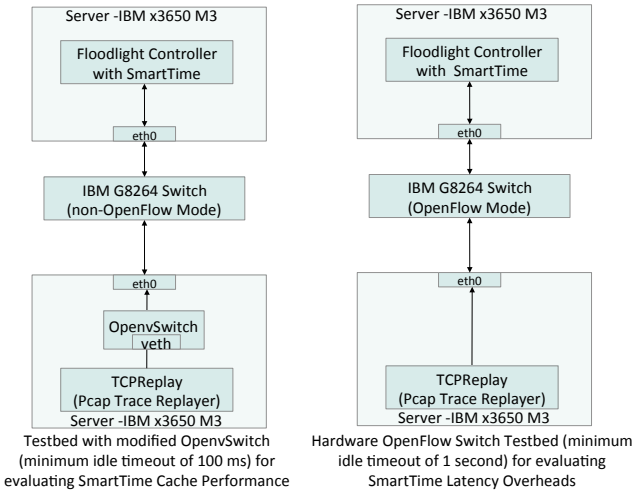


Figure 9: Testbeds for evaluating SmartTime

ment the SmartTime adaptive timeout algorithm (as shown in Figure 6), and additionally, if the TCAM utilization crosses a predetermined threshold level (95% in our current experiments), they evict an existing rule from TCAM, using a RANDOM criteria (Adaptive-R). We also experimented evicting using a FIFO order (Adaptive-F) with an additional check for a bad hold factor that ensures that only infrequent and short lived flows are removed (as shown in Figure 6). The results for adaptive policy with FIFO eviction were either very similar or slightly worse than adaptive policy with Random eviction.

In all our experiments, cache misses (drops or evictions in case of adaptive policy (Adaptive-R)) for each policy have been normalized with respect to the total cache misses (drops) across all the policies for that trace. We also plot the total cost of a policy as per Equation 4 in Section 3. Here, we assume that cost of installing a rule, is the same as cost of evicting a rule ($C_{evict}/C_{install} = 1$). Hence, total cost of a policy can be approximated by the sum of cache misses and drops (or evictions) of a policy divided by the sum of cache misses and drops (or evictions) across all policies for that trace. Total cost of a policy provides a better comparison across policies since a policy may result in low number of misses but may cause high drops/evictions (or vice versa).

Experiment 1 – Static Idle timeout (P1) vs Adaptive Policy (P3) for TCAM Size=750:

Figures 10(a), 10(b) and 10(c) show the cache misses, drops (proactive (or forced) evictions for adaptive policy) and total cost, respectively, for all the four traces: EDU1, EDU2, SMIA1, and SMIA2 for the static idle timeout policies (P1) and adaptive policy (P3). Following observations can be immediately made:

- No single static policy performs the best across all the four traces.Static-5000ms performs the best (lowest number of misses and lowest number of drops) for EDU1 trace, Static-500ms performs the best for EDU2 trace and Static-100ms performs the best for SMIA1 and SMIA2 traces.

- Adaptive policy (Adaptive-R) consistently outperformed the best performing static policy for three traces (Static-500ms for EDU2, Static-100ms for SMIA1 and SMIA2 traces) both in terms of misses and number of drops (evictions in case of Adaptive policy). Note that, in case of a static policy, a flow being dropped because of TCAM getting 100% utilized, results in all subsequent packets of that flow getting dropped. This can result in an inflation in number of drops for static policies.

- Adaptive policy (Adaptive-R) outperformed the best performing static policy for EDU1 (Static-5000ms) in terms of misses and underperformed in terms of drops (or evictions).

- In terms of total cost (Figure 10(c), Adaptive policy (Adaptive-R) outperformed the best performing static policy for EDU2 (Static-500ms), SMIA1 (Static-100ms) and SMIA2 (Static-100ms) by 25% to 58%. It outperformed the best performing static policy for EDU1 (Static-5000ms) by 4%.

Experiment 2 – Random eviction with static idle timeout (P2) vs Adaptive Policies (P3) for TCAM Size=750:

Figures 11(a), 11(b) and 11(c) show the cache misses, proactive (or forced) evictions and total cost, respectively, for all the four traces: EDU1, EDU2, SMIA1, and SMIA2 for Random eviction with static idle timeout policies (P2) and adaptive policy (P3). These experiments show the following trends:

- As was the case with static idle timeout policies, no single random eviction policy performs the best across all the four traces. In terms of total cost (Figure 11(b), random eviction with 5000ms static idle timeout (Random-5000ms) performs the best for EDU1 and EDU2 traces but Random-1000ms performs the best for SMIA1 and SMIA2 traces.

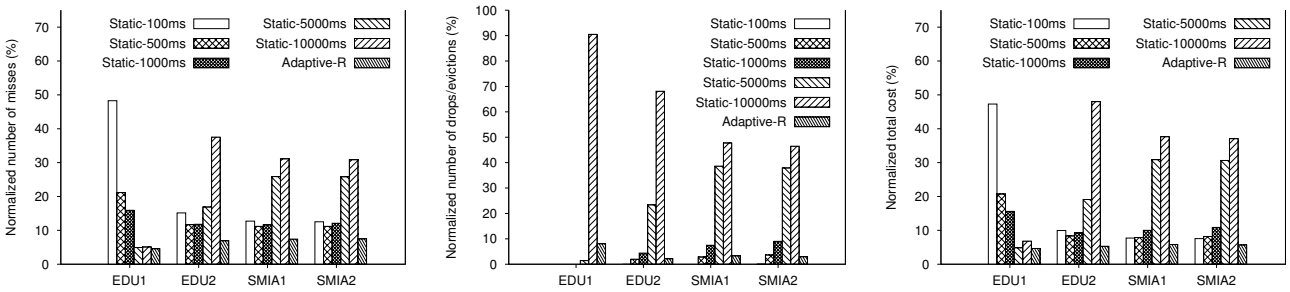
- In this experiment also, Adaptive policy (Adaptive-R) consistently outperformed the best performing random eviction policy for all the four traces by 2% to 15% in terms of total cost. The margin of outperformance is lower here (as compared to the static idle timeout experiment), and it highlights the benefits of proactively evicting flow rules at high TCAM utilization.

Experiment 3 – Static Idle timeout (P1) vs Adaptive Policy (P3) for TCAM Size=2000:

We now compare the effect of increasing the cache size from 750 to 2000. As discussed earlier, 2000 is a representative size for most TCAMs in current switches since Broadcom chipset used by most switches has a similar TCAM [15]. Figures 12(a), 12(b) and 12(c) show the cache misses, drops (proactive (or forced) evictions for adaptive policy) and total cost, respectively, for all the four traces: EDU1, EDU2, SMIA1, and SMIA2 for the static idle timeout policies (P1) and adaptive policy (P3). Most of the observations made earlier hold true even in this experiment:

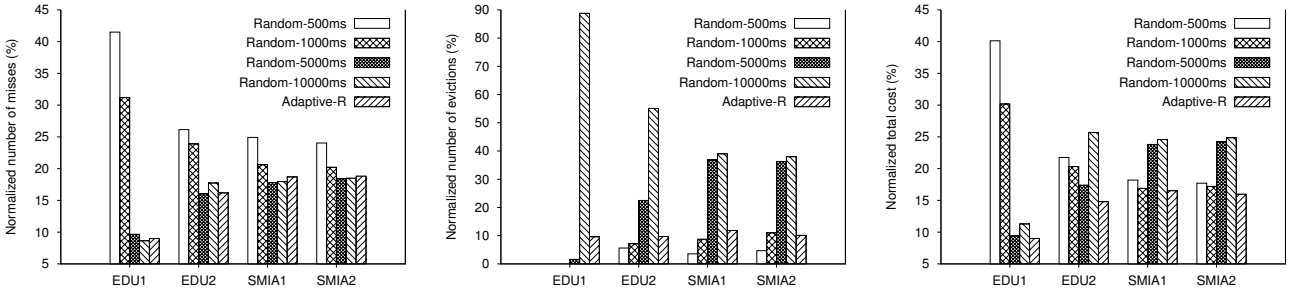
- No single static policy performs the best across all the four traces. In terms of total cost (Figure 10(c)) , Static-1000ms has the lowest total cost for EDU2, and Static-10000ms has the lowest total cost for EDU1, SMIA1 and SMIA2.

- Adaptive policy (Adaptive-R) outperformed the best performing static policy for EDU2 (Static-1000ms) by 52% and the best performing static policy for SMIA2 (Static-10000ms) by 16% in terms of total cost. It underperformed the best performing static policy for EDU1 (Static-10000ms) by 24% and the best performing static policy for SMIA1 (Static-10000 ms) by 6%. This is not entirely surprising since both EDU1 and SMIA1 do not have enough number of flows to exhaust the available TCAM space (2000). In the absence of any cache contention, a static policy with a high timeout will naturally perform the best. Even in these two cases,where the Adaptive-R policy underperformed the best performing static policy, it was still the second best performing policy across



(a) Misses (b) Drops/Evictions (c) Total Cost

Figure 10: Static idle timeout(P1) vs Adaptive Policies (P3) for Cache Size=750: Adaptive policy (Adaptive-R) outperforms the best performing static policy for EDU2(Static-500ms), SMIA1 and SMIA2 (Static-100ms) by up to 58% in terms of total cost. For EDU1 trace, it outperformed the best performing static policy (Static-5000ms) by 4%.



(a) Misses (b) Evictions (c) Total Cost

Figure 11: Random eviction with static idle timeout(P2) vs adaptive policies(P3) for Cache Size=750: Adaptive policy (Adaptive-R) outperforms the best performing random eviction policy for all the traces by up to 15% in terms of total cost.

all policies and utilized 25% less TCAM space for EDU1 and 70% less TCAM space for SMIA1.

Experiment 4 – Random eviction with static idle timeout (P2) vs Adaptive Policies (P3) for TCAM Size=2000:

Figures 13(a), 13(b) and 13(c) show the cache misses, proactive (or forced) evictions and total cost, respectively, for all the four traces: EDU1, EDU2, SMIA1, and SMIA2 for Random eviction with static idle timeout policies (P2) and adaptive policy (P3). These experiments show the following trends:

- In terms of total cost (Figure 13(b), random eviction with 10000ms static idle timeout (Random-10000ms) performs the best across all random eviction policies for all four traces.
- In this experiment, Adaptive policy (Adaptive-R) outperformed the best performing policy for EDU2 (Random-10000ms) by 19% in terms of total cost. It was the second best policy for EDU1, SMIA1 and SMIA2 traces, lagging slightly behind Random-10000ms by 24%,3% and 15% in terms of total cost, and utilizing 25%, 80% and 83% less TCAM space or evictions (in case of SMIA2), respectively.

Cache Performance Summary: In summary, the adaptive policy (Adaptive-R) consistently outperformed the best performing static idle timeout (P1) and random eviction (P2) policies across majority of all our experiments (in 67 experiments out of a total of 72 experiments- 4 traces, 9 static or random eviction policies, 2 cache sizes (750 and 2000)). In some cases, the outperformance was as high as 58% in terms of total cost. Only in cases when there was no cache contention (For cache size 2000, EDU1 and SMIA1 traces with static and random eviction policies, SMIA2 trace with random eviction policy), it slightly lagged behind the best performing static or random eviction policy. This is expected, since in the absence of any cache contention (and 0 drops), a high static idle timeout is likely to perform better than an adaptive policy. However, even in such cases, the underperformance was marginal (never worse than 24%) and Adaptive-R was the second best performing policy in terms of total cost, and utilized 25% to 80% less TCAM space.

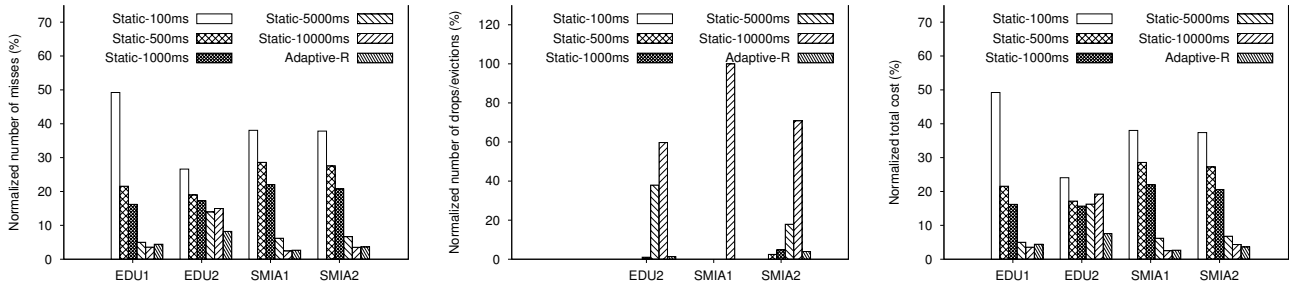
6.2 Evaluation of SmartTime Overheads

In order to evaluate the overheads of SmartTime policies, we performed three sets of experiments on the testbeds shown in Figure 9. In the first experiment, latency measurements using the ping utility were used to assess the overheads of SmartTime operations; in the second experiment, we studied the effect of increase in the number of flows in the TCAM on the performance of SmartTime; in the third experiment, we assess the overhead of SmartTime on real applications, with EDU2 trace.

Experiment 1 – Measurement of ping latencies:

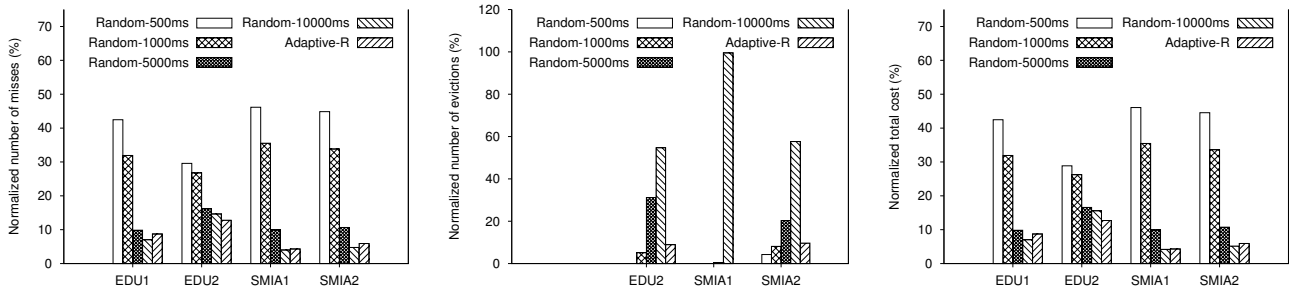
We used the ping utility to measure the latency in the network under five different scenarios. The experiments were performed on both the testbeds shown in Figure 9 with another server added as the ping destination. In each scenario, 2000 ping requests were generated at intervals of 100ms (for testbed 1, on the left of Figure 9) and 2 seconds (for testbed 2, on the right of Figure 9)[3] between the same source and destination, and their round trip times measured. In the description that follows, all chosen timeout values correspond to testbed 1; appropriate values were selected for testbed 2 to maintain the same hit-miss pattern as required by the scenario.

[3]This difference in ping intervals was required across the testbeds because the minimum idle timeout supported by the switch in testbed 2 was 1s; thus, generation of pings at an interval in the order of milliseconds would prevent us from capturing any misses



(a) Misses (b) Drops/Evictions (c) Total Cost

Figure 12: Static idle timeout(P1) vs Adaptive Policies (P3) for Cache Size=2000: Adaptive-R outperforms the best performing static policy for EDU2 (Static-1000ms) and SMIA2 (Static-10000ms) by up to 52% in terms of total cost. For EDU1 and SMIA1 traces that had no cache contention (and hence had 0 drops), Adaptive-R was the second best policy only slightly behind the best performing static policy (Static-10000ms) while utilizing 25% and 70% less TCAM space, respectively.



(a) Misses (b) Evictions (c) Total Cost

Figure 13: Random eviction with static idle timeout(P2) vs adaptive policies(P3) for Cache Size=2000: Adaptive policy (Adaptive-R) outperforms the best performing random eviction policy for EDU2 (Random-10000ms) by 19% in terms of total cost. It is the second best performing policy for EDU1, SMIA1 and SMIA2 traces, lagging slightly behind Random-10000ms in terms of total cost while utilizing 25%,80% and 83% less TCAM space or evictions (in case of SMIA2).

S1: No SmartTime, all flows resulting in hits – In this scenario, we ran SmartTime with a static idle timeout policy of 1s, after disabling all policy-related computations (thereby emulating a controller without SmartTime). Starting with an empty TCAM, the first ping request and response naturally result in misses leading to the installation of new flows; every subsequent packet results in a cache hit.

S2: No SmartTime, all flows resulting in misses – We used the same modified version of SmartTime as in the previous scenario, but with a static idle timeout policy of 10ms. Thus, each ping request (and response) results in a miss, the inter-packet arrival times being greater than the idle timeout for flows.

S3: SmartTime with Adaptive-R policy, all flows resulting in hits –In this scenario, SmartTime was run with Adaptive-R policy with a minimum idle timeout of 1s. As in scenario (S1), only the first ping request (and corresponding response) result in the installation of new flows; all subsequent packets result in hits.

S4: SmartTime with Adaptive-R policy, all flows resulting in misses – In this scenario, we ran SmartTime with Adaptive-R policy, with a constant idle timeout of 10ms for all flows, while keeping all SmartTime computations intact. We achieved this by overriding the computed timeout values with a constant 10ms during flow installations, only for the purpose of expediting the experiment, without affecting the performance of SmartTime. As in scenario (S2), all packets result in misses.

S5: SmartTime with Adaptive-R policy, all flows resulting in evictions – Unlike the previous scenarios, we first populated the

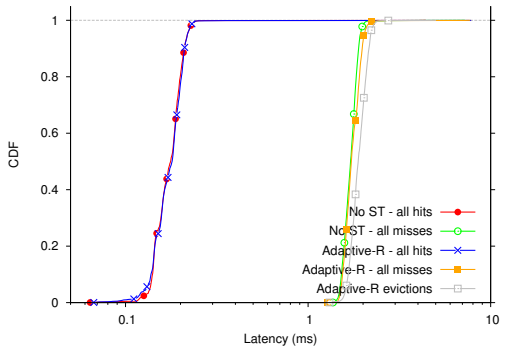


Figure 14: CDF of ping packet latencies with OVS testbed shown on the left hand side of Figure 9

TCAM completely with randomly generated permanent flows. The idle timeout for the ping flows, however, was maintained constant at 10ms as in scenario (S4). Moreover, after each ping, we additionally installed two permanent flows in the TCAM, to occupy the space left vacant after the ping request and response flows timed out. This ensures that the TCAM is full whenever a ping packet reaches the switch, and an installation is always preceded by an eviction.

Figure 14 shows the CDF of the ping latencies (in log scale) under the different scenarios on the OVS testbed shown on the left hand side of Figure 9 (testbed 1), while Figure 15 shows the results on the hardware switch testbed on the right hand side of Figure 9 (testbed 2). The average latency of the packets (after discarding the top and bottom 10%) are presented in Table 2 for both sets of results. As expected, negligible difference in latency was seen be-

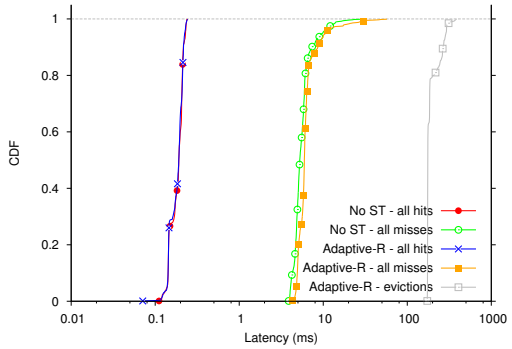


Figure 15: CDF of ping packet latencies with hardware testbed shown on the right hand side of Figure 9

S.No.	Scenario	Avg. latency (ms)	
		Testbed 1	Testbed 2
1	No SmartTime – all hits	0.174	0.185
2	No SmartTime – all misses	1.712	5.399
3	Adaptive-R – all hits	0.176	0.184
4	Adaptive-R – all misses	1.749	5.987
5	Adaptive-R – evictions	1.897	186.6

Table 2: Avg. latency of ping packets

tween scenarios (S1) and (S3), where all packets result in cache hits, and the controller does not participate. To calculate the overheads of SmartTime in the case of a new flow installation (without eviction), we consider scenarios (S2) and (S4); the observed difference in latency being the result of SmartTime computations involved in the installation of two new flows – ping request and the corresponding response. The overhead in this case is (1.749 - 1.712)/1.712 = 0.022 or 2.2% for testbed 1, while it is (5.987 - 5.399)/5.399 = 0.109 or 10.9% for testbed 2. Note that difference in absolute values of overhead between the two testbeds is a consequence of the difference in specifications of the servers on which the controller is running. For the case in which an installation is preceded by an eviction, we consider scenarios (S2) and (S5), the observed overhead being (1.897 - 1.712)/1.712 = 0.108 or 10.8% for testbed 1. The high latency observed in testbed 2, we suspect, is due to anomalous behavior in the hardware switch and is currently under investigation. [4]

It is important to note here that the difference in latency between scenarios (S4) and (S5) cannot be attributed completely to flow eviction. As is evident from the results of Experiment 2 (described subsequently), the cost of flow installations by SmartTime increases with increase in the number of flows installed in the TCAM; and while the TCAM was always empty whenever a new flow arrived in scenario (S4), in scenario (S5) it remains full.

Additionally, from the results of the experiment on testbed 1, the cost of flow installation $C_{install}$ and the overhead of eviction C_{evict} can be computed thus: $C_{install} = 1.749$ ms, (from the observations of scenario (4)), and $C_{evict} = 1.897 - 1.749 = 0.148$ ms (from scenarios (4) and (5)). Also, $C_{\text{evict}}/C_{\text{install}} = 0.085$; this is in stark contrast with our assumption in Section 6.1, where this factor was generously assumed to be 1. Note that the exact value of this factor will differ across setups; the smaller this factor, the lesser the contribution of eviction overheads towards the total cost.

[4]We suspect that the high overhead of evictions seen in testbed 2 is a result of inefficiencies in the eviction algorithm implemented in the firmware, and not an artifact of SmartTime. We have confirmed this by studying Wireshark packet captures at the controller while observing the latencies between OpenFlow packets during our experiments. Moreover, had this been a result of our SmartTime policies, similar behavior would have been observed on testbed 1.

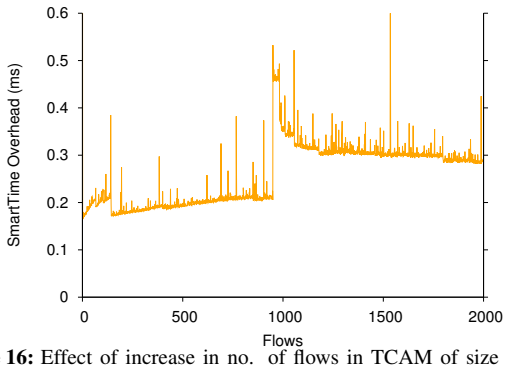


Figure 16: Effect of increase in no. of flows in TCAM of size 1000 on SmartTime performance

Experiment 2 – Effect of increase in number of flows on controller performance:

In this experiment, the overheads of SmartTime were computed from within the controller, using system time measurements. We injected 2000 randomly generated flows into a TCAM of size 1000, with an Adaptive-R policy having a threshold of 95%, modified to install permanent flows (while maintaining all policy related computations). The results are presented in Figure 16. We observed a gradual increase in the time taken for flow installations (without evictions) until the time that the TCAM threshold was reached. This is to be expected as SmartTime maintains a history of observed flows to aid its decisions. This increase, however, is small, and of the order of 0.1ms. Any new flows beyond the TCAM threshold involve an additional eviction operation. In steady state, the cost of eviction followed by installation stabilizes, and remains unaffected by further increase in the number of observed flows.

Experiment 3 – Application performance on a hardware testbed:

In order to evaluate the overhead of SmartTime policies on application overall performance on a hardware testbed, we performed experiments on the hardware testbed shown on the right hand side of Figure 9. We executed the netperf [5] benchmark application while tcpreplay continued replaying the captured traces. Trace replay was required in order to ensure that TCAM on the hardware switch was either full or close to being full, and almost all netperf executions resulted a cache miss and a possible eviction in case of proactive eviction policies. We used netperf in the request-response mode (TCP RR test [6]) to capture increase in flow setup latencies. We executed netperf for 2 seconds duration and each run was repeated after 15 seconds as tcpreplay continued replaying EDU2 trace. This ensured that each netperf run resulted in a cache miss (since the maximum idle timeout used by SmartTime is 10 seconds). Average latency and throughput results of these runs is given in Figure 17. Almost all policies perform nearly equally with minor (less than 1%) differences in latency or throughput numbers. This confirms that SmartTime does not result in any significant overheads.

Experiment 4 – CPU overheads at the controller:

We collected the CPU utilization at the controller (running on a 12 core server) as we conducted the cache performance experiments of Section 6.1 on the OVS testbed shown on the left hand side of Figure 9. We compared the CPU utilization for best performing static idle timeout policy and random eviction policy for all the four traces with the CPU utilization for Adaptive-R policy. The overall CPU utilization at the controller was never more than 1% for any of our experiments. Average CPU utilization at the

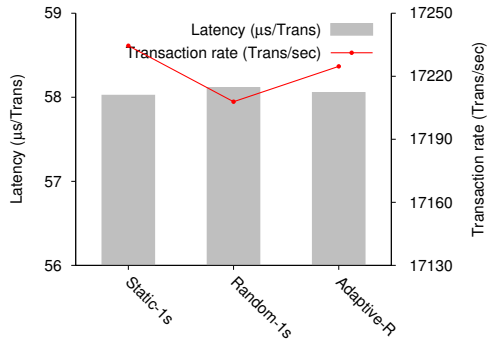


Figure 17: Netperf experiments for evaluating SmartTime effect on application performance: Almost all policies perform nearly equally with minor (less than 1%) difference

controller was marginally higher for Adaptive-R policy while the Max CPU utilization was lower as compared to best performing static and random eviction policies. This could be due to the fact that Adaptive-R policy avoids bursts of misses and evictions (which would result in high Max CPU), while continuously computing the best idle timeout for each TCAM miss (which results in higher average CPU). Since the overall CPU utilization was less than 1%, we omit these results due to lack of space.

7. CONCLUSION

In this paper, we presented SmartTime: a first real implementation of an OpenFlow controller system that combines adaptive idle timeouts for flow rules with a proactive eviction strategy based on current TCAM utilization level and can be deployed in current OpenFlow networks.

We formulated the problem of minimizing the total cost of a TCAM miss in an OpenFlow network. We analyzed real data center packet traces and designed our adaptive heuristic based on some key observations. Based on our observation that many flows in the network never repeat, we recommend a smaller minimum timeout value for OpenFlow flow rules (in the range of 10-100 milliseconds as compared to the current minimum timeout value of 1 second).

We validated SmartTime by replaying four real data center packet traces for two representative cache sizes and compare it with multiple static idle timeout policies and random eviction policies. In all our experiments, SmartTime adaptive policy was either the best performing or second best performing policy across all traces and cache sizes. In 67 out of 72 experiments, SmartTime adaptive policy outperformed the best performing static idle timeout policy or random eviction policy by up to 58% in terms of total cost.

Currently, we are involved in enhancing SmartTime adaptive policy through several important additions. First, we are fine tuning our adaptive policy so that it can react to inactive flows faster by taking into account feedback on number of packets or bytes that have matched an expired flow rule. Second, we are working on an automated framework for automatically selecting some of the tuning parameters (initial or the min idle timeout, max idle timeout, rate of timeout increase) based on observed network parameters. Finally, we are also working towards deploying and validating SmartTime in production OpenFlow networks.

8. REFERENCES

[1] Data Set for IMC 2010 Data Center Measurement. `http://pages.cs.wisc.edu/~tbenson/IMC10_Data.html`.

[2] Data Set from The Swedish Defence Research Agency (FOI) Information Warfare Lab. `http://www.foi.se/en/Our-Knowledge/Information-Security-and-Communication/Information-Security/Lab-resources/CRATE/`.

[3] Floodlight: A Java-based OpenFlow Controller. http://www.projectfloodlight.org/floodlight/.

[4] IBM RackSwitch G8264 Command Reference: Pages 103-107. `http://www-01.ibm.com/support/docview.wss?uid=isg3T7000600&aid=1`.

[5] NetPerf Benchmark. http://www.netperf.org/netperf/NetperfPage.html.

[6] NetPerf TCP RR test. `http://www.netperf.org/netperf/training/Netperf.html#0.2.2Z141Z1.SUJSTF.8R2DBD.R`.

[7] Open vSwitch - An Open Virtual Switch. http://www.openvswitch.org.

[8] OpenDaylight: A Linux Foundation Collaborative Project. http://www.opendaylight.org.

[9] TCPReplay: Pcap replay tool. http://tcpreplay.synfin.net/wiki/tcpreplay.

[10] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *ACM SIGCOMM*, 2011.

[11] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI*, 2003.

[12] N. Gude et al. NOX: Towards an Operating System for Networks. In *ACM SIGCOMM CCR*, July 2008.

[13] A. Iyer, V. Mann, and N. Samineni. SwitchReduce: Reducing switch state and controller involvement in OpenFlow networks. In *IFIP Networking*, 2013.

[14] K. Kannan and S. Banerjee. Flowmaster: Early eviction of dead flow on sdn switches. In *ICDCN*, 2014.

[15] G. Lu, C. Guo, Y. Li, Z. Zhou, T. Yuan, H. Wu, Y. Xiong, R. Gao, and Y. Zhang. Serverswitch: A programmable and high performance platform for data center networks. In *NSDI*, 2011.

[16] V. Mann, A. Gupta, P. Dutta, A. Vishnoi, P. Bhattacharya, R. Poddar, and A. Iyer. Remedy: Network-aware Steady State VM Management for Data Centers. In *IFIP Networking*, 2012.

[17] V. Mann, A. Kumar, P. Dutta, and S. Kalyanaraman. VMFlow: Leveraging VM Mobility to Reduce Network Power Costs in Data Centers. In *IFIP Networking*, 2011.

[18] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Syst. J.*, 9(2):78–117, June 1970.

[19] N. McKeon et al. OpenFlow: Enabling Innovation in Campus Networks. In *ACM SIGCOMM CCR*, 2008.

[20] B. Ryu, D. Cheney, and H. Braun. Internet flow characterization: Adaptive timeout strategy and statistical modeling. In *Passive and Active Measurement Workshop (PAM)*, 2001.

[21] S.Kandula, S.Sengupta, A.Greenberg, and P.Patel. The Nature of DataCenter Traffic: Measurement and Analysis. In *IMC*, 2009.

[22] T.Benson, A. Akella, and D.Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[23] A. Zarek, Y. Ganjali, and D. Lie. Openflow timeouts demystified. In *MSc Thesis for Department of Computer Science, University of Toronto*, 2012.